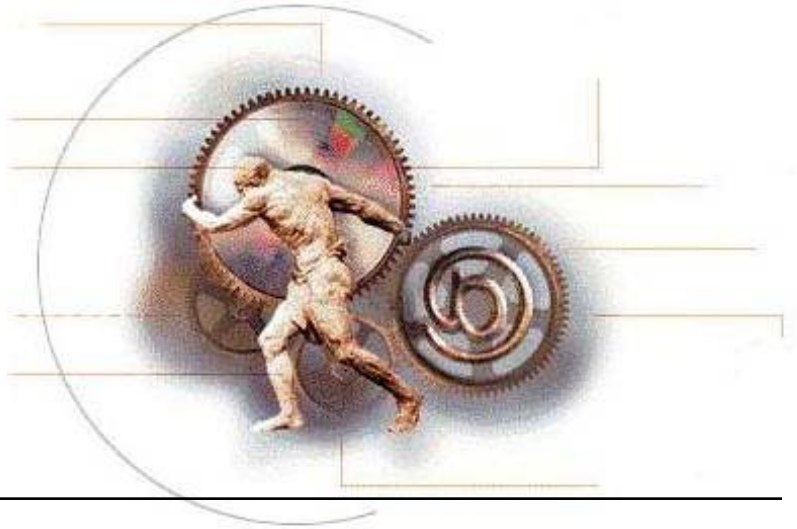


Course 8

回溯、分枝與限制

Backtracking, Branch-and-Bound

國立聯合大學 資訊管理學系 陳士杰老師

Outlines

- 本章重點
  - 求解Optimization Problems
  - Backtracking vs. Branch and Bound
  - Backtracking
  - Branch and Bound

求解Optimization Problems

- 若以暴力演算法來求算最佳化問題，對於有n個輸入項目的最佳化問題 (X1, X2, ..., Xn):
  - 有些被歸類爲 "**部份集合(Subset)**" 問題，則會有**2^n種**可能的情況
    - 如：**部份集合之和 (Sum of Subset)問題**、**0/1背包問題**...等
  - 有些被歸類爲 "**排列(Permutation)**"問題，則會有**n!種**可能的情況
    - 如：**N皇后(N-Queen)問題**、**旅行銷售員問題(Traveling Salesman Problem; TSP)**、**漢米爾頓迴路(Hamiltonian Circuits)問題**、**圖形著色(Graph-Coloring)問題**...等
  - 上述問題若以暴力法來解，皆屬指數複雜度的問題，若可採用最佳化原則，通常可以將這一些問題的複雜度由指數複雜度降爲多項式複雜度。

- **Dynamic Programming** 和 **Greedy Approach**所能處理的最佳化問題需滿足最佳化原則：
  - **最佳化原則(Principle of Optimality)**: 當一個問題存在著某個最佳解，則表示在此最佳解中，也必存著該問題之所有子問題的最佳解
  - 此類問題可利用 “貪婪法則” 或 “動態規劃” 來設計演算法
- 然而，並不是所有求最佳化的問題都合乎最佳化原則，此時就只能用其它的方法求解了。

Backtracking vs. Branch and Bound

- 對於**具有限制的最佳化問題**，除了可以採用 "貪婪法則" 或 "動態規劃" 來設計演算法則之外，若問題不具有 "最佳化原則"時，可考慮採用**回溯(Backtracking)**或**分枝與限制(Branch and Bound)**之解題策略。
  - 這兩種解題策略均是將問題的所有可能解答，表示成一個稱爲**狀態空間樹 (State Space Tree)** 的樹狀結構。接著，
    - 回溯策略採用 "**深先搜尋法**" (Depth-First Search; DFS) 對狀態空間樹中每一個節點進行檢查
    - 分枝與限制策略採用 "**廣先搜尋法**" (Breadth-First Search; BFS) 對狀態空間樹中每一個節點進行檢查
  - 上述的兩個策略，皆透過 "**邊界函數**" (Bounding Function) 來刪除一些不必要的子樹搜尋動作，以提昇搜尋效率。

解答空間 (Solution Space)

- 每個問題通常都可爲其定義一個**解答空間 (Solution Space)**:

  S = { (X1, X2, ..., Xn) ; (X1, X2, ..., Xn)爲滿足問題的所有解**}**

  - 這個空間必須至少包含該問題的一個解答，而這個解答也可能就是一個最佳解。

- 以具有n個商品的0/1背包問題來說，其解答空間是由2^n個可能解答所構成；每一個可能解爲n個 0 或 1所構成之集合，這個集合表示 "對所有商品Xi分別指派0和1的可能方法"

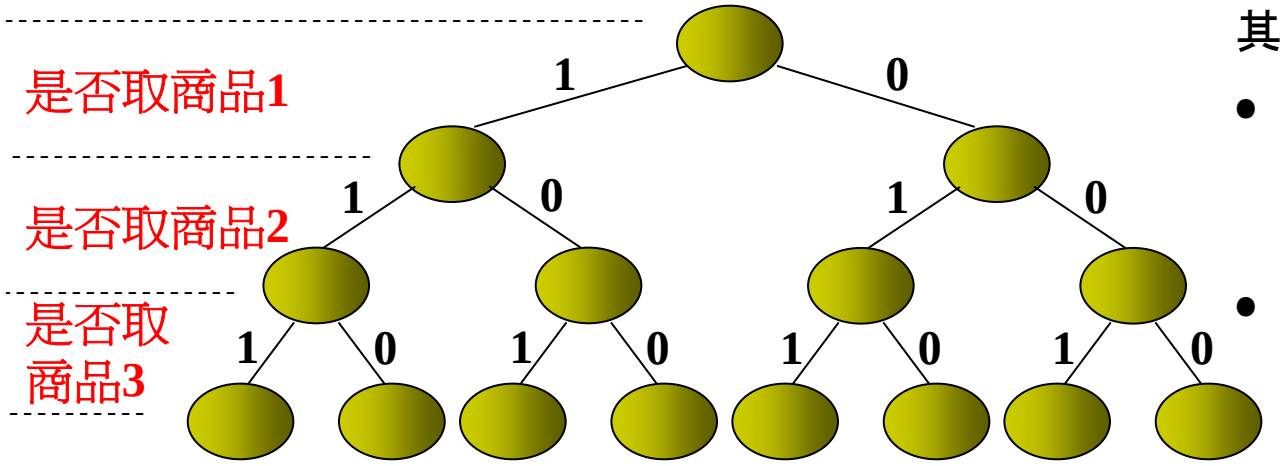
  - 若有3個商品 (n=3)，其0/1背包問題之解答空間共有$2^n = 2^3 = 8$個可能解:

    S = {(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)}

  - 爲了方便搜尋解答空間，我們可以將解答空間組織化，其中最典型的組織方式是樹狀結構，又稱**狀態空間樹 (State Space Tree)**

狀態空間樹 (State Space Tree)

- 某些問題的解答，可以產生樹狀結構來列舉所有可能的答案組合。任何一條從樹根節點到葉節點的路徑就是一個可能的答案，這個樹狀結構也因此稱爲**狀態空間樹(State Space Tree)**。
- 下圖是3個商品的0/1背包問題之狀態空間樹：

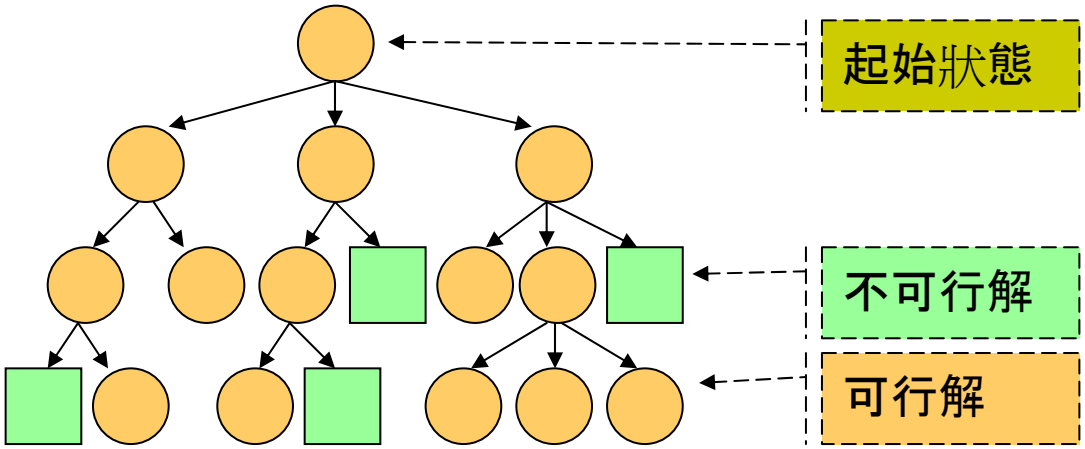


其中：

- 從樹根到葉節點的每一條路徑，皆為解答空間中的一個元素
- 有$2^3 = 8$個可能解(或葉節點)

- 由上圖可知，如果是有n個輸入項目的**排列問題**或是**部份集合問題**，則狀態空間樹所有可能的狀態個數(即：葉節點個數)分別爲$n!$或是2^n。
- 因此，回溯和分枝與限制的設計策略中，會加入使用**邊界函數**來決定是否需要繼續搜尋後續的狀態空間，以去除不必要的子樹搜尋。若：
  - 當搜尋到 "不可行" 的節點 (即：課本所指的沒前途(nonpromising)節點) 時，則不用再去搜尋該節點以下之所有分枝節點。
    - 此即爲修剪 (Pruning)
  - 當搜尋到 "可行" 的節點 (即：課本所指的有前途(promising)節點) 時，則可以再續繼往下搜尋該節點以下之分枝節點。
  - 可以得到**修剪過的狀態空間樹 (Pruned State Space Tree)**。

- 一個修剪過的狀態空間樹：



- 回溯和分枝與限制這兩種解題策略算是暴力法的改良版，是藉由狀態空間樹，對所有可能的狀態進行有系統地搜尋，雖然整體的時間複雜度沒有降低，但是藉由省去部份不必要的步驟，可以提升系統的效能，所以還是比利用暴力法好一些。

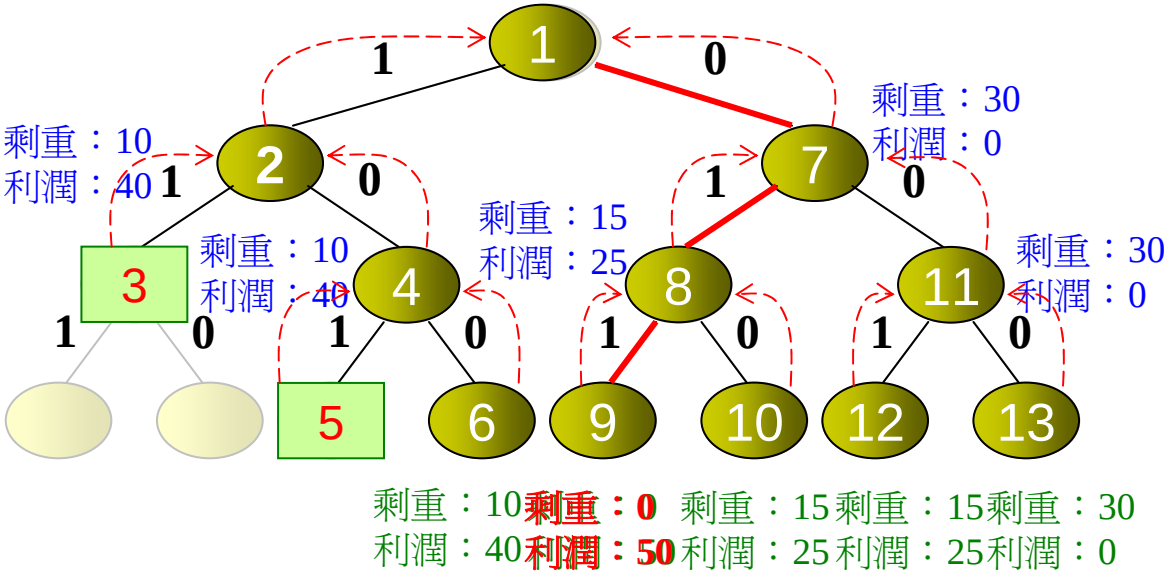
Backtracking (回溯)

- 採用"深先搜尋法" (Depth-First Search; DFS) 對狀態空間樹中每一個節點進行檢查。
  - 爲遞迴的應用概念，因此可利用Stack保存走訪過程中間所走過的點。
- 有3個不可分割的商品，其重量與價值分別如下。若背包容量爲30公斤 (C=30)，請利用回溯策略找出最佳解答。

Item	重量 (W)	價值(P)
1	20	$40
2	15	$25
3	15	$25

Item	重量 (W)	價值(P)
1	20	$40
2	15	$25
3	15	$25

- 三個商品的0/1背包問題的狀態空間樹如下：
  - 邊界函數爲 C ≥ Σ(wi×Xi), 其中：
    - wi是商品i的重量 (整數變數)
    - Xi是商品i是否有被拿取 (布林變數)



- 搜尋過程(利潤與背包剩餘重量不說明)：
  - 由樹根先出發，若要拿商品1，則移到節點2
  - 接著，由節點2出發，若要拿商品2，則移到節點3。但是因爲拿商品1又拿商品2，超出背包之負重，因此爲不可行之解，退回到上層節點(即：節點2)。
  - 再由節點2出發，此時選擇不拿商品2，則移到節點4。
  - 從節點4出發，若要拿商品3，則移到節點5。但是拿商品1又拿商品3，超出背包之負重，因此爲不可行之解，退回到上層節點(即：節點4)。
  - 再由節點4出發，此時選擇不拿商品3，則移到節點6。節點6爲一個可行解，但由於該節點沒有子節點，因此回到上層節點(即：節點4)。
  - 節點4已無未搜尋之節點，因此回到上層節點(即：節點2)。
  - 節點2已無未搜尋之節點，因此回到上層節點(即：節點1)。

- 再由樹根先出發，此時選擇不拿商品1，則移到節點7
- 接著，由節點7出發，若要拿商品2，則移到節點8。
- 再由節點8出發，此時選擇拿商品3，則移到節點9。節點9爲一個可行解，但由於該節點沒有子節點，因此回到上層節點(即：節點8)。
- 再由節點8出發，此時選擇不拿商品2，則移到節點10。節點10爲一個可行解，但由於該節點沒有子節點，因此回到上層節點(即：節點8)。
- 節點8已無未搜尋之節點，因此回到上層節點(即：節點7)。
- 從節點7出發，此時選擇不拿商品2，則移到節點11。
- 從節點11出發，若要拿商品3，則移到節點12。節點12爲一個可行解，但由於該節點沒有子節點，因此回到上層節點(即：節點11)。
- 再由節點11出發，此時選擇不拿商品3，則移到節點13。節點13爲一個可行解，但由於該節點沒有子節點，因此回到上層節點(即：節點11)。
- 節點11與節點7已無未搜尋節點，因此回到最上層節點。

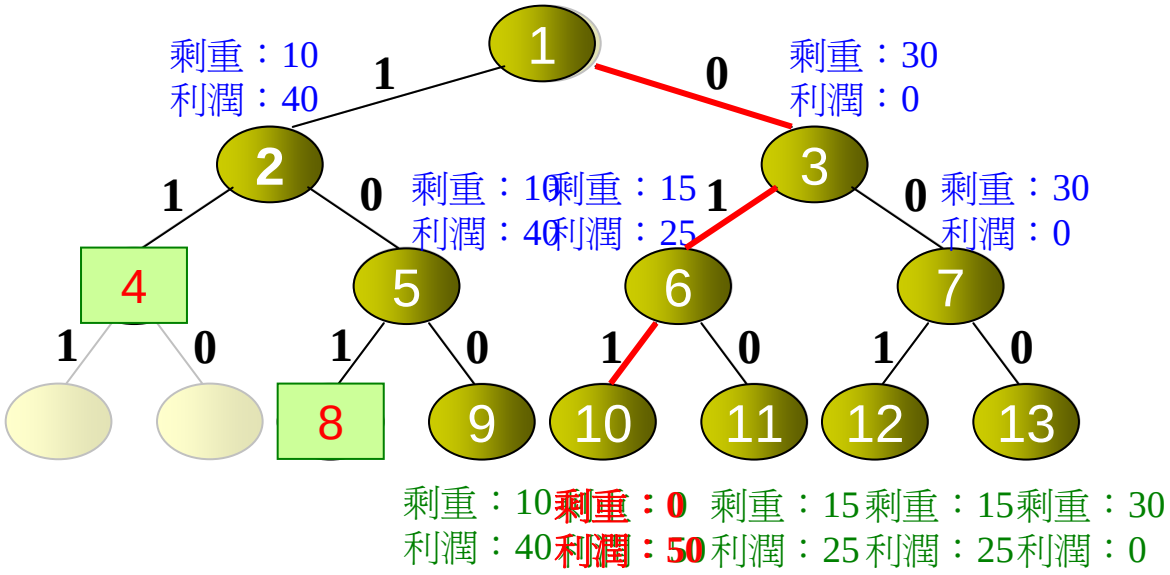
Branch and Bound (分枝與限制)

- 採用"廣先搜尋法" (Breadth-First Search; DFS) 對狀態空間樹中每一個節點進行檢查。
  - 爲迴圈的應用概念，因此可利用Queue保存走訪過程中間所走過的點。
- 有3個不可分割的商品，其重量與價值分別如下。若背包容量爲30公斤 (C=30)，請利用分枝與限制策略找出最佳解答。

Item	重量 (W)	價值(P)
1	20	$40
2	15	$25
3	15	$25

Item	重量 (W)	價值(P)
1	20	$40
2	15	$25
3	15	$25

- 三個商品的0/1背包問題的狀態空間樹如下：
  - 邊界函數爲 C ≥ Σ(wi×Xi), 其中：
    - wi是商品i的重量 (整數變數)
    - Xi是商品i是否有被拿取 (布林變數)

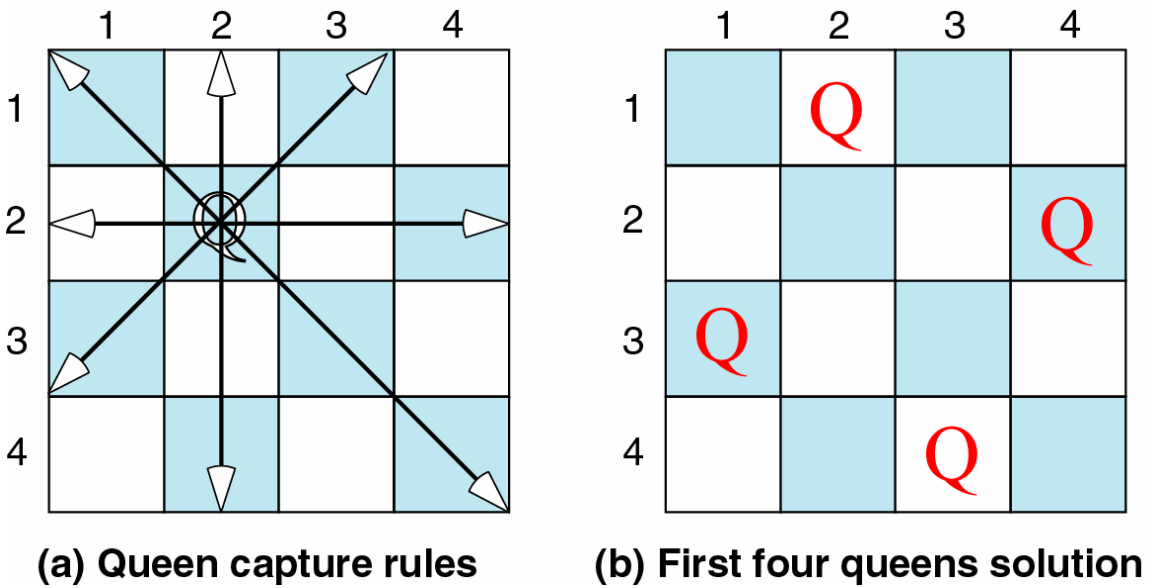


- 由於是利用佇列 (Queue) 做爲輔助工具。因此，最先被放入的節點將最先被處理
- 搜尋過程：
  - 由樹根先出發，將根節點移動一步就可以到達之所有未被搜尋過的子節點依序存入佇列中。
  - 從佇列中移出一個節點，當做出發節點，並將此節點移動一步就可以到達之所有未被搜尋過的子節點依序存入佇列中。
  - 不斷執行步驟2直到所有節點執行完畢。執行中須隨時注意是否有超出背包負重。

補　充

補 1: n-皇后問題

- 所謂n皇后問題是指 “如何將n顆西洋棋中的皇后棋子擺放在一個具有n列n行的棋盤中，但是這n顆棋子彼此不會吃掉對方，也就是這n顆皇后棋子彼此不允許在同一列、同一行或是同一個對角線”。
- 以下是四皇后問題示例：

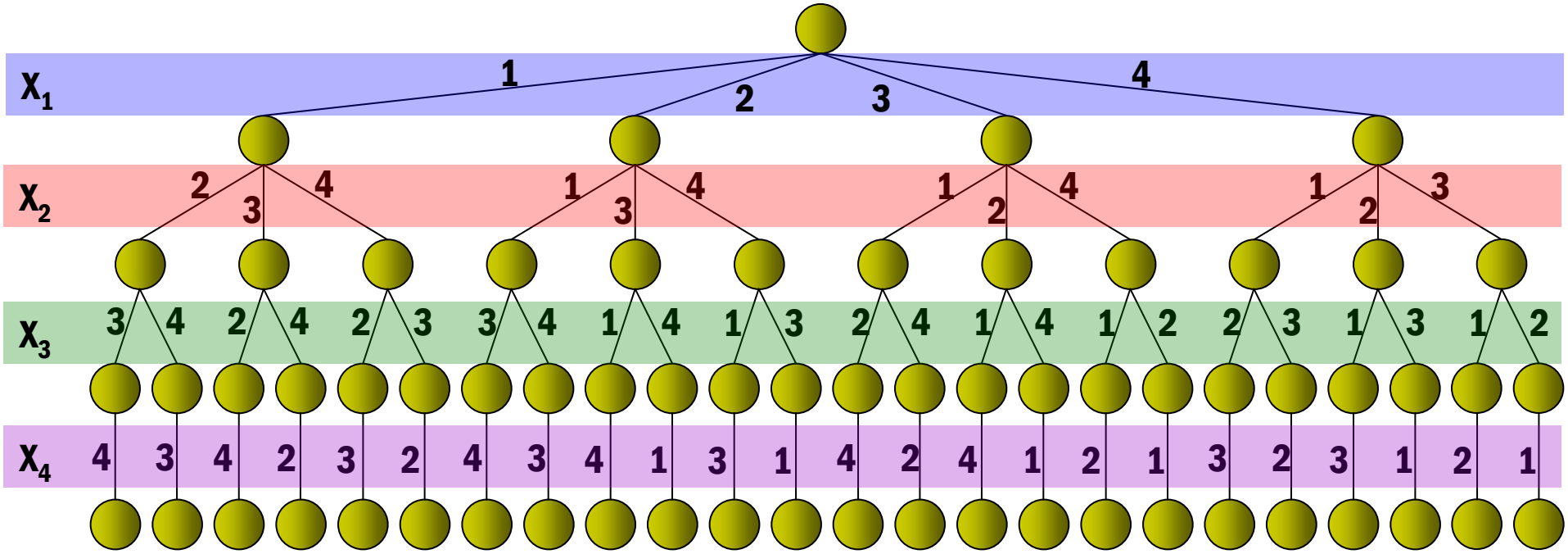


(a) Queen capture rules (b) First four queens solution

- 利用暴力式的直覺作法，可能有下列兩種解題方法：
  - 作法 **1**：
    - **n**個皇后需放置於不同列上，並且檢查每個皇后在其所屬之列上的哪一個行才是其應放置的位置。
    - 第一列可擺放的位置有**n**個，第二列可擺放的位置有**n**個，第三列可擺放的位置有**n**個，**...**，第**n**列可擺放的位置有**n**個。
    - $n \times n \times n \times \ldots \times n = n^n$
  - 作法 **2**：
    - **n**個皇后需放置於不同列、不同行之位置上。且除了與先前擺放之皇后的同行位置外，需檢查每個皇后在其所屬之列上的哪一個行才是其應放置的位置。
    - 第一列可擺放的位置有**n**個，第二列可擺放的位置有**n-1**個，第三列可擺放的位置有**n-2**個，**...**，第**n**列可擺放的位置有**1**個。
    - $n \times (n-1) \times (n-2) \times \ldots \times 1 = n!$
- 以暴力法來解決**n**皇后問題，似乎採用作法 **2**較爲明智**!!**

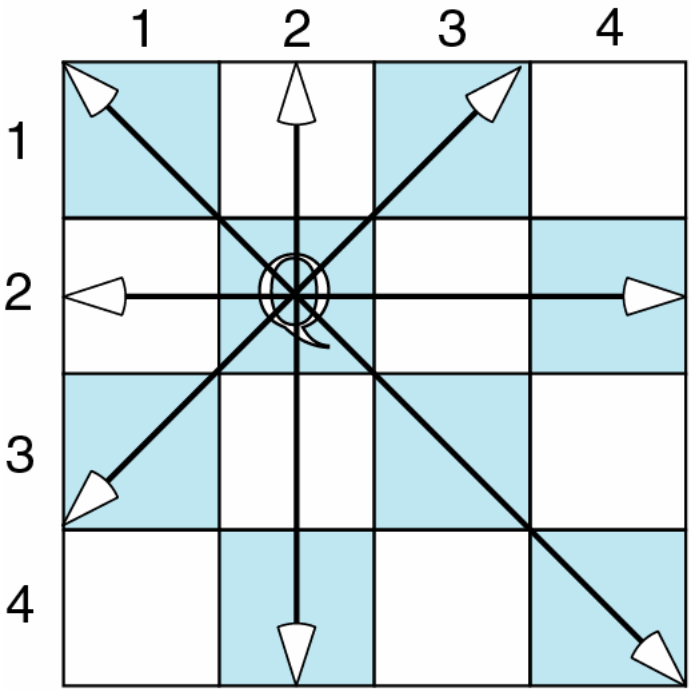
- 以四皇后問題爲例，上述兩種暴力作法之解答空間爲：
  - **[作法 1] $4^4 = 4 \times 4 \times 4 \times 4 = 256$** 個可能解
  - **[作法 2] $4! = 4 \times 3 \times 2 \times 1 = 24$**個可能解
- 因此，**n-**皇后問題屬於排列問題 **(**採用暴力法**)**。

- 四皇后問題的狀態空間樹：
  - 假設x_i表示在第i列的皇后棋子所在之行位置 (課本上是以col(i)表示x_i)，其中i = 1~n。



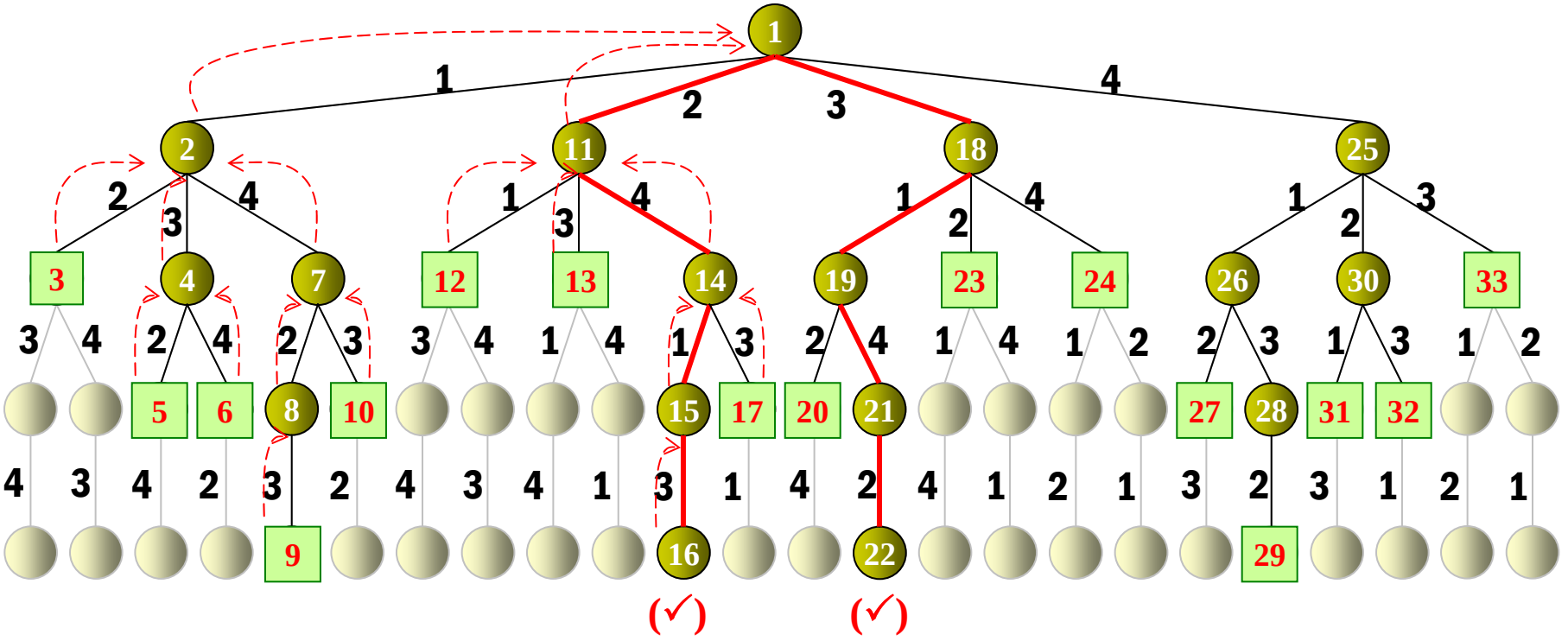
如何判斷兩個皇后間是否互吃

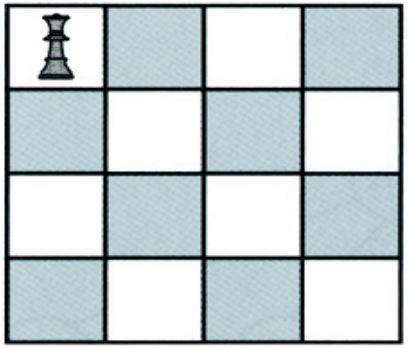
- 假設Q1皇后所在的位置是(a, x_a)，Q2皇后所在的位置是(b, x_b)，下列位置皆會造成兩皇后互吃。
  - 如果$b = a$，則表示Q1皇后和Q2皇后在同一列
  - 如果$x_b = x_a$，則表示Q1皇后和Q2皇后在同一行
  - 如果$(x_b - x_a) = b - a$，則表示Q1皇后和Q2皇后皆在右斜45°線
  - 如果$(x_b - x_a) = -(b - a)$，則表示Q1皇后和Q2皇后皆在左斜45°線



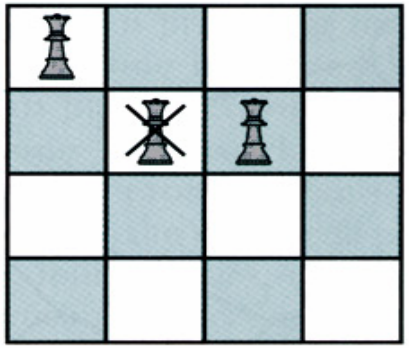
利用回溯法解4皇后問題

- 4皇后問題的狀態空間樹如下：
  - 邊界函數爲前面四個互吃之判斷條件。

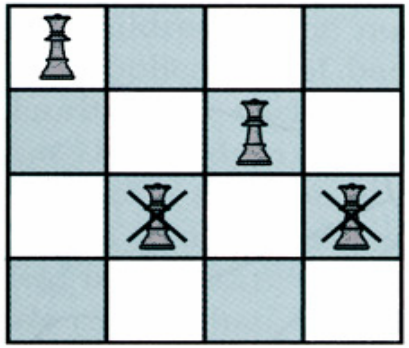




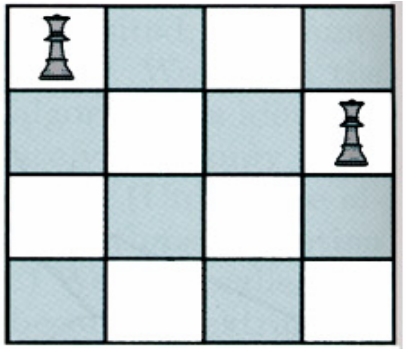
(a)



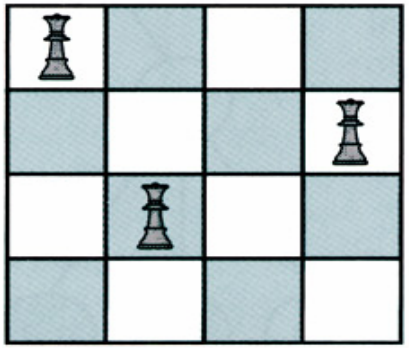
(b)



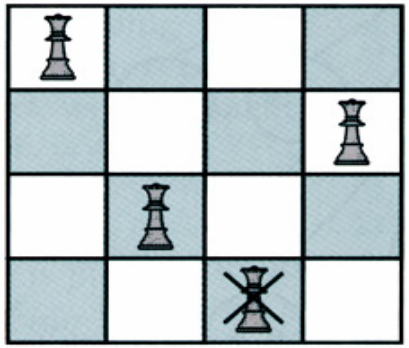
(c)



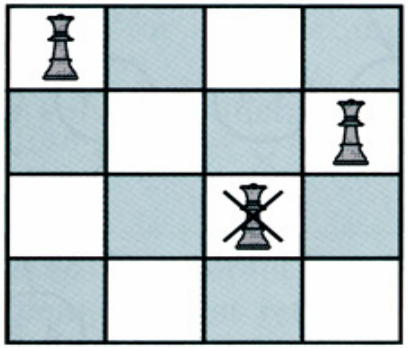
(d)



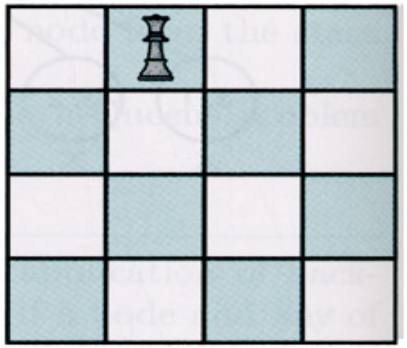
(e)



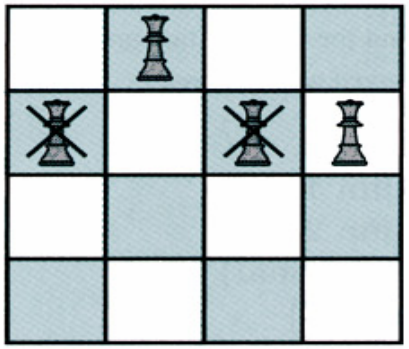
(f)



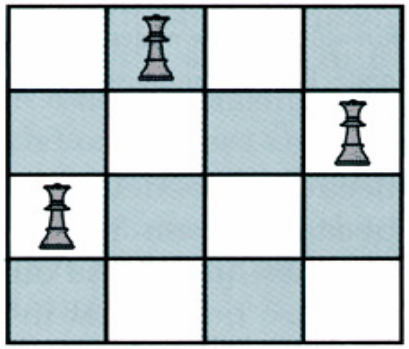
(g)



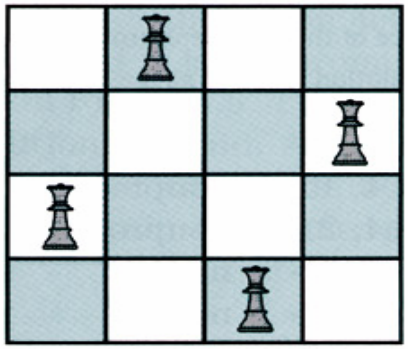
(h)



(i)

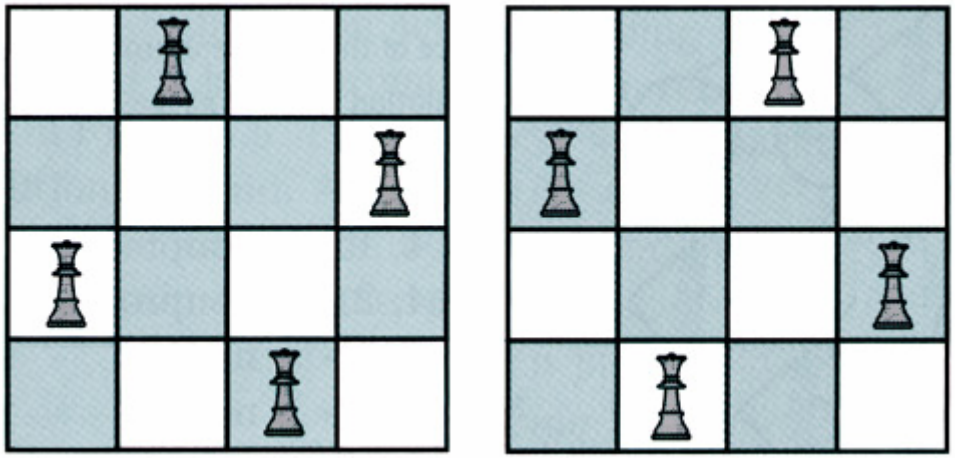


(j)



(k)

- 四皇后問題的解答



Algorithm 5.1: The Backtracking Algorithm for the *n*-Queens Problem

Problem: Position *n* queens on a chessboard so that no two are in the same row, column, or diagonal.

Inputs: positive integer *n*.

Outputs: all possible ways *n* queens can be placed on an $n \times n$ chessboard so that no two queens threaten each other. Each output consists of an array of integers *col* indexed from 1 to n, were *col[i]* is the column where the queen in the *i*th row is placed.

```
void queens (index i)
    {
  index j;

  if (promising (i))
         if (i == n)
             cout << col [1] through col [n];
         else
            for (j = 1; j <= n; j++){      // See if queen in
                    col [i + 1] = j;       // (i + 1) st row can be
                    queens (i + 1);        // positioned in each of
                                           // the n columns.
                }
    }
bool promising (index i)
    {
index k;
bool switch;
k = 1;
switch = true;                                    // Check if any queen threatens
while (k < i && switch){  // queen in the ith row.
    if (col [i] == col [k] || abs(col [i] - col [k])== i-k)
           switch = false;
    k++;
    }
    return switch;
}
```

※ 註：
- col[i] = X_i